Benchmarking GNU Radio Kernels and Multi-Processor Scheduling

Nathan West, Doug Geiger, George Scheets

January 14, 2013

1 Introduction

The growth of Software Defined Radio (SDR) using general purpose processors (GPPs) brings a new engineering decision of processor selection to radio design. In this study we are concerned with comparing the performance of an SDR toolkit called GNU Radio on different processors. Properly selecting a processor for a radio application will depend on the application; however, a generic list of benchmarks would include

- Floating Point OPerations per second (FLOPs) for common routines on multiple-processors
- time to complete common math/type conversions
- system latency

Knowledge of an application could then be paired with benchmark comparisons of potential processors to make an informed decision on the best processor for size, weight, power or cost constrained applications.

This study focuses on the first two items and leaves latency measurements as another project. [7] worked on latency measurements between an earlier version of GNU Radio and an Ettus USRP with a USB connection. Tallying FLOPs for FIR and FFT blocks in series and parallel covers two commonly used processes and tests the ability of a processor to multitask parallel operations and the ability of a processor to use multiple threads [3]. The second benchmark is common math and type conversions. GNU Radio provides the Vector Optimized Library of Kernels (VOLK) library that selects the best Single Instruction Multiple Data (SIMD) architecture for a given processor and operation to speed up computation [10, 12]. An example of this might be in a dot product, a common signal processing operation, which requires a point-by-point multiplication of two vectors followed by the sum of the products. In many cases, taking advantage of a SIMD architecture yields faster processing time than the equivalent C-style loop [10, 14, 13]. Examples of SIMD architectures are the SSE instructions common in Intel and AMD platforms, while on the ARM architecture the NEON instruction set provides a similar set of SIMD instructions. VOLK, which has been

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
14 JAN 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Benchmarking GNU Radio Kernels and Multi-Processor Scheduling	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for Applied Research in Artificial Intelligence,4555 Overlook Ave., SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
New England Workshop for Software Defined Radio (NEWSDR'13), Worcester, MA, 17 May 2013.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**23**	

included in GNU Radio since December 2010, attempts to easily take advantage of SIMD instruction sets on all processors without having the application programmer be concerned about using SIMD instructions [10, 11, 12]. The first attempt to benchmark VOLK performance, in February 2012, released tools to time specific math and type-conversion kernels [11]. The current work combines the efforts to benchmark VOLK and multiprocessor scheduling of FFTs and FIRs to a single suite with the intention of comparing potential processors.

Since the interest is in comparing hardware performance, a build framework, called Open Embedded (OE), is used to install Linux with GNU Radio on the different machines. OE is a collection of tools and metadata that can cross-compile a complete Linux system with any applications pre-installed [15]. The OE metadata is separated into layers based on the use and maintainability of the software being built [4]. Most of the required system tools are hosted in a layer called *openembedded-core* (*oe-core*); the kernel, and machine description for real hardware comes from a board support package layer; the applications and development tools come from *meta-openembedded* [4]. There are other layers that describe the file system layout, software to be installed and corresponding versions, and a build system that comes from distribution layers such as Angstrom or Poky [15, 9].

2 Methodology

This study uses a range of hardware to test the effectiveness of benchmarking. The processors tested so far include

- Intel i7 with hyper threading
- Intel Atom with hyper threading
- AMD E350 APU, comparable to Atom
- ARM Cortex A8 running on a Gumstix Overo on an Ettus USRP E110

The general testing procedure consists of

- Build Linux, GNU Radio, and file system for target machine
- Run benchmarking scripts with VOLK enabled
- Run benchmarking scripts with VOLK disabled
- Optionally run oprofile with desired application

The results from each type of benchmark can be compared with the application profile to select the appropriate processor.

2.1 Open Embedded

In order to compare the differences in hardware, the software should be as controlled as reasonably possible. Using OE to build the Linux kernels and GNU Radio provides a reasonably fair platform to run benchmarking tools from [8]. OE is also a reasonable choice for creating the benchmark system since the main interest is in size, weight, and power constrained systems, which will generally be embedded. We use the Poky distribution that is included via the *meta-yocto* layer of OE, and define our custom image which is based on *core-image-minimal*. Our new image installs GNU Radio, which is provided in *meta-openembedded*, and brings in the integrated benchmarking suite.

2.2 Existing Benchmarking Code

We modified the existing benchmarking tools to store data in a consistent format that is text-based to avoid dependencies on databases and make results more portable. The plotting tools for the multi-processor scheduler benchmarking (FFT/FIR arrays) were changed to use MatPlotLib so that the whole suite uses the same tools [5]. The plotting for VOLK benchmarking was also changed to retrieve data from the new format and to create consistent coloring of the results for easier comparisons.

2.3 Application Profiling

For benchmarking an application we use oprofile, which samples the CPU either after a certain number of CPU events occur or after a regular time specified, to return the percentage of running time used in various functions [8, 2]. It is important that debug symbols be included in the target image so that applications can be profiled [8]. Since each function call, for example a single VOLK kernel, becomes a compiler symbol, by profiling a running GNU Radio application the functions that use the most time can be easily identified.

3 Results

3.1 Open Embedded

Getting a Poky distribution build with GNU Radio to boot on x86_64 (Intel Atom and AMD E350) and ARM (TI OMAP 3530) created several initial problems since the primary meta-oe development is done by Angstrom developers [1] which focuses primarily on ARM architectures. The primary issue was with the device manager and systemd; specifically, building GNU Radio brings in systemd through a chain of dependencies. To solve the booting issues on x86_64 and build GNU Radio we used the version of udev provided by *oe-core* and use BBMASK variables to mask out everything in the GNU Radio dependency chain that would lead to systemd or *meta-oe* udev versions being built. Falling back on older and more stable *oe-core* tools where necessary resulted in a Linux

image with GNU Radio that could boot and run the exact same software on the AMD-based Gumstix board (Ettus E110), the Intel Atom, and the AMD E350 APU.

3.2 Multi-Processor Scheduling

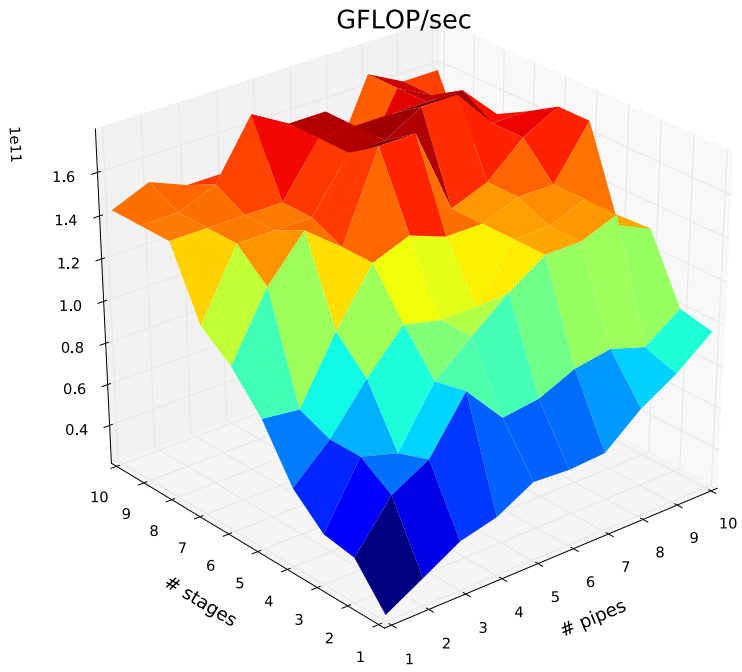

Figure 1: GFLOPs per second through an FFT array on an Intel i7.

Example output from multi-processor scheduling tests on an Intel i7 are shown in Figures 1, 2, 3, 4. The big performance differences come from using VOLK while doing parallel FFTs. On the FFT plots, Figures 1 and 2, comparing the gradient in the direction of pipes verses stages yields an interesting difference between generic kernels and using VOLK. Using generic kernels adding stages causes a larger increase to FLOPs/s compared to adding pipes; in contrast with VOLK enabled adding pipes causes an equivalent increase in FLOPs/s as adding

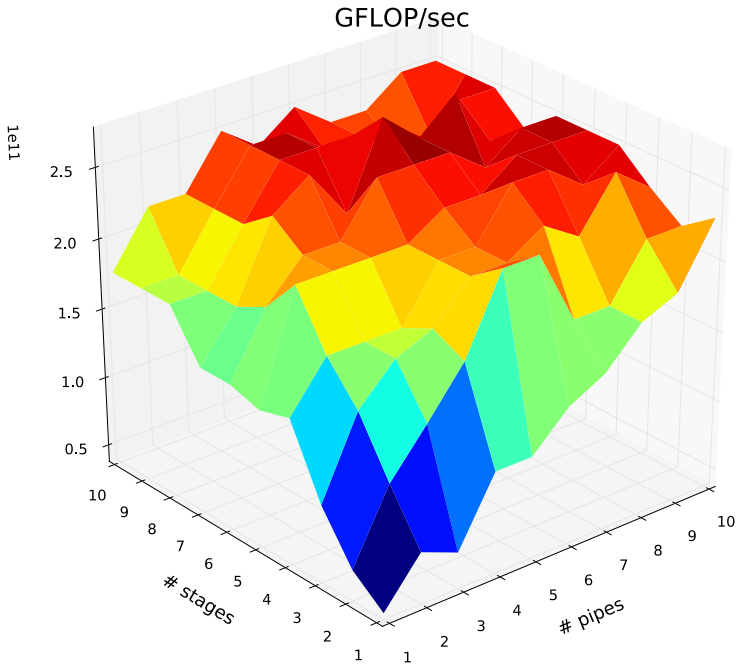

Figure 2: GFLOPs per second through an FFT array with VOLK enabled on an Intel i7.

stages does. Every processor will have a peak in the measured FLOPs where adding more pipes and stages will not yield an increase in FLOPs and past which the measured FLOPs will decrease, likely due to the increased need to store and fetch samples from memory.

The FIR filter results do not seem affected by VOLK kernels being used as opposed to the generic kernels. This is likely because FIR filter blocks have already been hand-tuned to use SIMD instructions without VOLK [6].

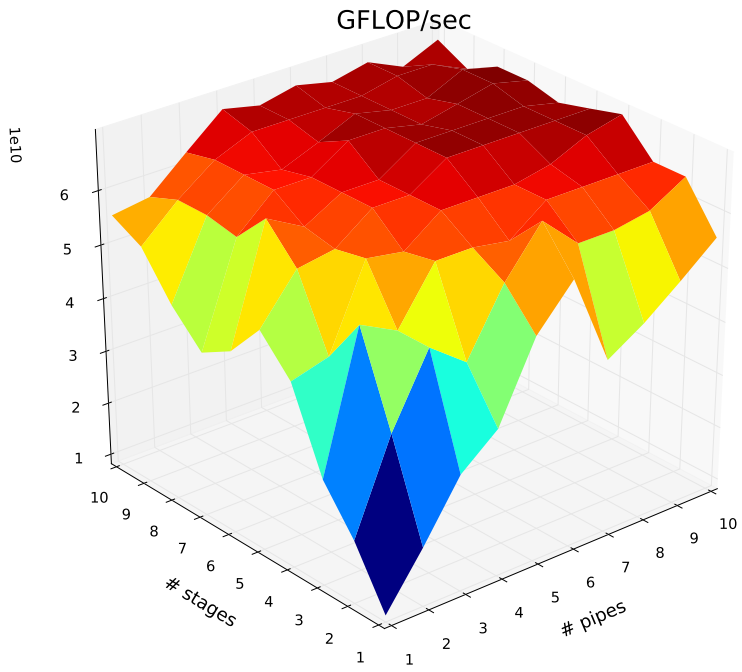

Figure 3: GFLOPs per second through an FIR filter array on an Intel i7.

3.3 VOLK kernels

Figures 5 and 6 show the results of benchmarking VOLK math operations implemented as stand-alone GNU Radio blocks using generic kernels and with VOLK optimizations, respectively. Type conversions with generic and VOLK

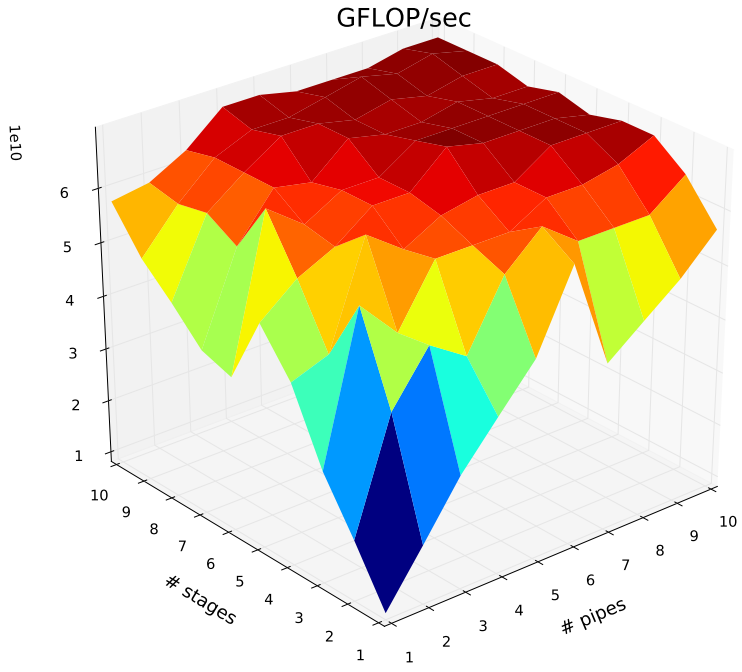

Figure 4: GFLOPs per second through an FIR filter array with VOLK enabled on an Intel i7.

improvements are shown in Figures 7 and 8. The height of each bar in these four graphs is the total time to repeat the named operation 1 billion times; the black bar shows one standard deviation of those 1 billion measurements.

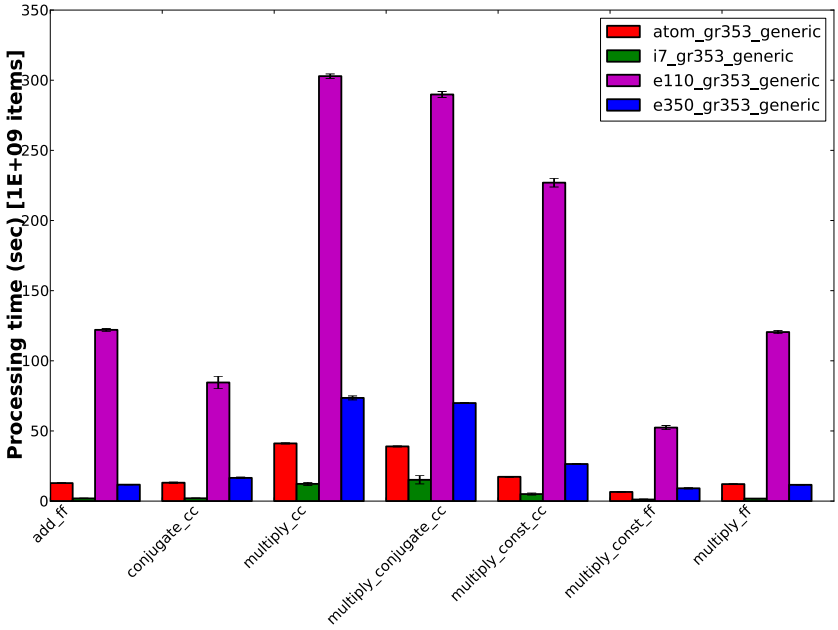

Figure 5: VOLK math results using a generic VOLK kernel for different processors.

The Intel i7 is obviously much faster than other processors, and the E110 is obviously much slower. The E110 sees no improvement from VOLK across all benchmarks because ARM processors use the NEON architecture that is not well supported in VOLK yet. Some highlights from the VOLK results are that nearly all instructions on the x86 processors do see improvement from generic kernels. It is also clear that some processors have more performance gain from VOLK than others. For example comparing the multiply_cc, which executes a complex multiply and outputs a complex result, in Figure 5 the Atom is clearly faster than the E350 using generic kernels. Using VOLK, Figure 6 shows that the E350 is now slightly faster than the Atom. Similar results appear Figures 5 and 6 with multiply_conjugate_cc, complex_to_mag in Figures 7 and 8.

Some VOLK kernels are actually on par with or slower than the generic kernels. Since this occurs mostly on the simpler instructions such as add_ff and multiply_const_ff the likely cause is compilers and processors are already fine-tuned to do these instructions efficiently.

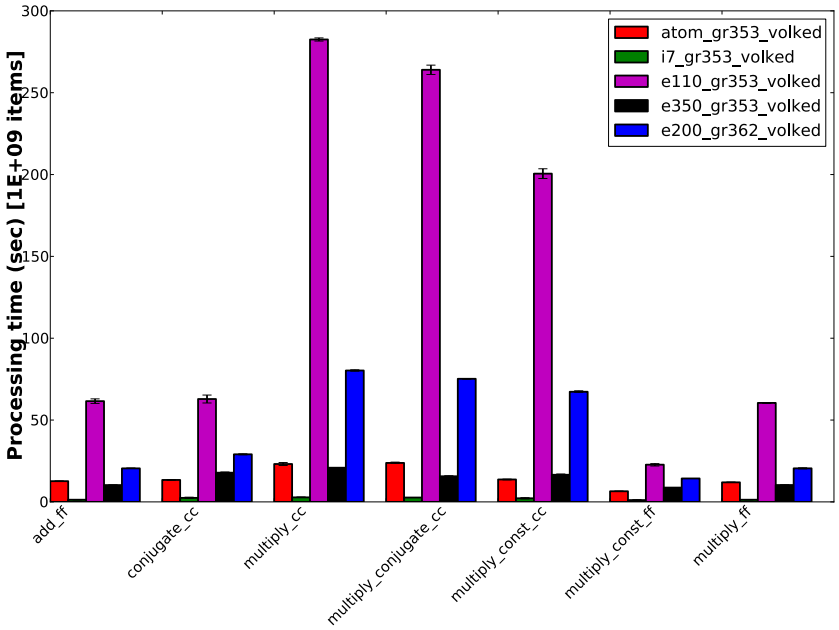

Figure 6: VOLK math results using VOLK kernels for different processors.

4 Conclusion

Multi-processor scheduling and speed of vector math will play an important role in processor selection for a software radio application. Being able to pick a processor that matches size, weight, and power constraints that match desired specifications requires knowledge of processor performance. By integrating and enhancing existing tools and working through the bugs to run GNU Radio through a Poky distribution build of OE we have introduced a benchmarking platform which can assist in choosing the best platform for embedded software radios. We have also introduced benchmarking results for a small set of potentially suitable processors for software radio.

References

[1] Contributors to openembedded/meta-oe, October 2012. https://github.com/openembedded/meta-oe/graphs/contributors.

[2] Oprofile, August 2012. http://oprofile.sourceforge.net/about/.

[3] Eric Blossom. GNU Radio - MP Scheduler Performance. GNU Radio, 9 edition, July 2008. http://gnuradio.org/redmine/projects/gnuradio/wiki/MPSchedulerPerformance.

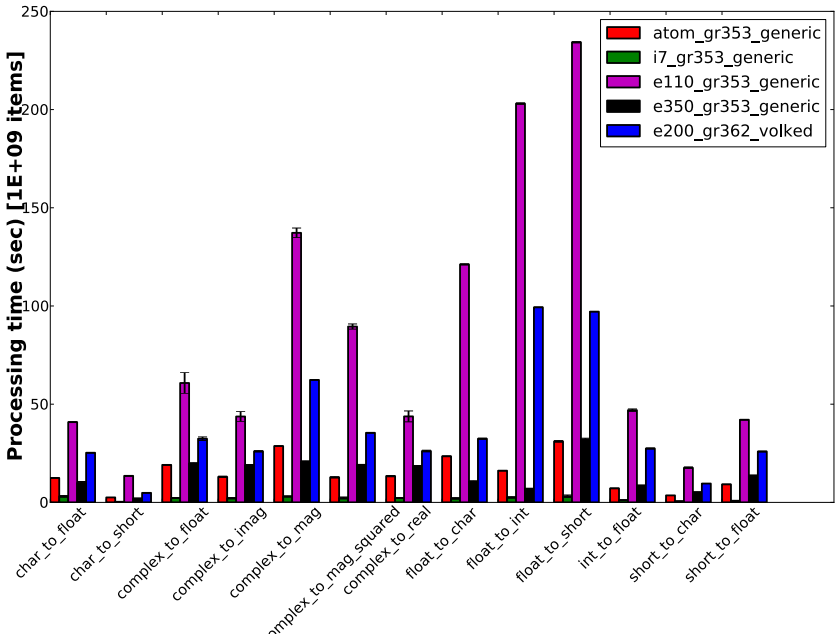

Figure 7: VOLK type conversion results using a generic VOLK kernel for different processors.

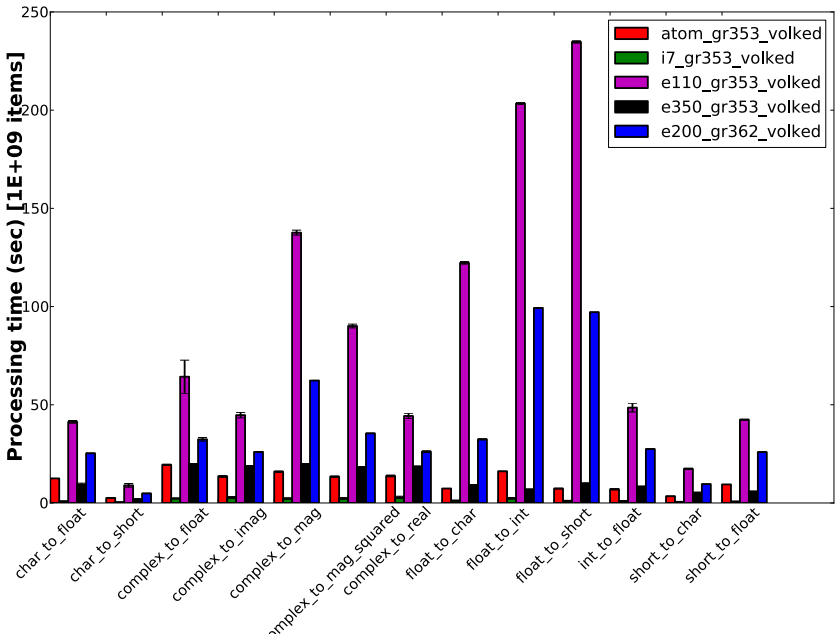

Figure 8: VOLK type conversion results using VOLK kernels for different processors.

[4] Paul Eggleton and Lloyd Chang. Openembedded-core, September 2012. http://www.openembedded.org/wiki/OpenEmbedded-Core.

[5] John Hunter, Darren Dale, Eric Firing, and Michael Droettboom. matplotlib, November 2012. http://matplotlib.org/contents.html.

[6] Sean Nowlan. Discuss-gnuradio, November 2011. http://lists.gnu.org/archive/html/discuss-gnuradio/2011-11/msg00151.html.

[7] George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, and Peter Steenkiste. Enabling mac protocol implementations in software-defined radios. NSDI'09 Proceedings of the 6th USENIX symposium on Networked systems design and implementation, 2009.

[8] Richard Purdie. The Yocto Project Reference Manual. Linux Foundation, July 2012. http://www.yoctoproject.org/docs/current/poky-ref-manual/poky-ref-manual.html.

[9] Scott Rifenbark. The Yocto Project Development Manual. Intel and Linux Foundation, July 2012. http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html.

[10] Thomas Rondeau. Volk: Vector-optimized library of kernels. blog, December 2010. http://www.trondeau.com/blog/2010/12/11/volk-vector-optimized-library-of-kernels.html.

[11] Thomas Rondeau. Volk benchmarking. blog, February 2012. http://www.trondeau.com/blog/2012/2/17/volk-benchmarking.html.

[12] Tom Rondeau and Dimitrios Symeonidis. GNU Radio - VOLK. GNU Radio, 11 edition, March 2012. http://gnuradio.org/redmine/projects/gnuradio/wiki/Volk.

[13] Kees van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman, and Matthias Weiss. Vector processing as an enabler for software-defined radio in handheld devices. EURASIP J. Appl. Signal Process., 2005:2613–2625, January 2005.

[14] M. Woh, Yuan Lin, Sangwon Seo, T. Mudge, and S. Mahlke. Analyzing the scalability of simd for the next generation software defined radio. In Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on, pages 5388 –5391, 31 2008-april 4 2008.

[15] Yocto Project. Openembedded Core, October 2012. http://www.yoctoproject.org/projects/openembedded-core.

Listing 1: Benchmark control script.

```
#!/bin/sh

# 1st parameter is processor description
# 2nd parameter is gnuradio version
ARCH=$1
GR_VERSION=$2
FNAME=$1"_"$2"_"
export PYTHONPATH=utils/
volk_profile
 mp-sched/run_synthetic_fft.py -m 10 -D 'synth_fft.txt' -L $FNAME"
     fft_volked"
 mp-sched/run_synthetic_fir.py -m 10 -D 'synth_fir.txt' -L $FNAME"
     fir_volked"
volk/volk_math.py -L $FNAME"volked" -D volk_math.db --all
volk/volk_types.py -L $FNAME"volked" -D volk_types.db --all
# Change all architectures in volk_profile to generic
# with sed magic
echo 'old volk_config moved to ~/.volk/volk_config.volked'
echo 'generic volk_config being generated'
mv ~/.volk/volk_config ~/.volk/volk_config.volked
sed 's/\(orc\|neon\|ss\)[a-z0-9_]*/generic/' ~/.volk/volk_config.volked
     > ~/.volk/volk_config
mp-sched/run_synthetic_fft.py -m 10 -D 'synth_fft.txt' -L $FNAME"
    fft_generic"
mp-sched/run_synthetic_fir.py -m 10 -D 'synth_fir.txt' -L $FNAME"
    fir_generic.raw"
volk/volk_math.py -L $FNAME"generic" -D volk_math.db --all
volk/volk_types.py -L $FNAME"generic" -D volk_types.db --all
```

Listing 2: Modified helper functions for testing.

```
#!/usr/bin/python
#!/usr/bin/env python

from gnuradio import gr
import math,sys,os,time,re,pickle

try:
    import numpy
except ImportError:
    sys.stderr.write("Unable␣to␣import␣Numpy\n")
    sys.exit(1)

# tables can have names with any letter, number, underscore, period, or
     dash
table_name_chars = "[a-zA-Z0-9_.␣-]"


def common_args(parser):
    parser.add_argument('-D', '--database', type=str, required=True,
                        help='Database␣(pickled␣file)␣to␣rw␣results')
    parser.add_argument('--listtables',
                        default=False, action='store_true',
                        help='print␣a␣list␣of␣tables␣in␣the␣database␣
                            file')
    return parser
```

```
def create_connection(fname):
    '''
    return a file object. If it's not createx
    '''
    try:
        return open(fname,'r+')
    except IOError:
        return open(fname,'w+')

def new_table(conn, tablename):
    '''
    Create a new "table" of sorts for the results. Each table
    should be for a different architecture/machine/type. It's
    best to keep these names unique, but there's no checker
    planned for that.
    You should run list_tables first to make sure that you
    aren't duplicating a table. I don't know what will happen
    '''
    conn.seek(0,os.SEEK_END) # go to end
    # command is the string to write to file
    cmd = "\n<{0}>\n".format(tablename)
    conn.write(cmd)


def insert_results(conn, res):
    '''
    Insert results that are apparently dictionary values into
    the table. Since this is a text file we pickle the list.
    '''
    # because mp-sched was designed to be two independent
    # programs (why oh why oh why?! - fix this later)
    # appending to a file is the "best" way to insert results
    conn.seek(0,2) # go to the last byte in the file
    # the whole dictionary gets pickled
    conn.write("$newdatarow$")
    pickle.dump(res,conn)

def list_tables(conn):
    '''
    return a list of all tables in the database
    '''
    conn.seek(0)
    fstring = conn.read()
    # match as few characters as possible inside angle brackets
    table_names = re.findall("<("+table_name_chars+"*?)>", fstring)
    return table_names

def get_results(conn, tname):
    '''
    gets all results in tname. tname should match an arch+gr version
    No testing is done on originality of tname
    '''
    pickle_string = '(\([a-zA-Z0-9\'\n]*s\.)?'
    conn.seek(0)
    fstring = conn.read()
    # match the < to begin a table
```

```
    tables = re.split('(<.*>)?',fstring)
    for i,table in enumerate(tables):
        if table.startswith('<'+tname+'>'):
            res = list()
            entries = re.split("\$newdatarow\$", tables[i+1])
            for row in entries:
                try:
                    rdata = pickle.loads(row)
                except:
                    # yikes! I hope that wasn't data!
                    print
                else:
                    rdata
                    res.append(rdata)
            return res

def close_connection(conn):
    conn.close()
```

Listing 3: Modified MatPlotLib for VOLK results plotting.

```
#!/usr/bin/python
#!/usr/bin/env python

import sys, math
import argparse
from common_test_funcs import *
import collections

try:
    import matplotlib
    import matplotlib.pyplot as plt
except ImportError:
    sys.stderr.write("Could not import Matplotlib (http://matplotlib.
        sourceforge.net/)\n")
    sys.exit(1)

def main():
    desc='Plot Volk performance results from a SQLite database. ' + \
        'Run one of the volk tests first (e.g, volk_math.py)'
    parser = argparse.ArgumentParser(description=desc)
#    parser.add_argument('-D', '--database', type=str,
#                        default='volk_results.db',
#                        help='Database file to read data from [default
    : %(default)s]')
    parser.add_argument('-E', '--errorbars',
                        action='store_true', default=False,
                        help='Show error bars (1 standard dev.)')
    parser.add_argument('-P', '--plot', type=str,
                        choices=['mean', 'min', 'max'],
                        default='mean',
                        help='Set the type of plot to produce [default:
                            %(default)s]')
    parser.add_argument('-%', '--percent', type=str,
                        default=None, metavar="table",
                        help='Show percent difference to the given type
                            [default: %(default)s]')
    parser.add_argument('-T', '--tables', type=str, nargs='*',
```

```
                    default=None,
                    help='select the tables to plot')
parser.add_argument('-o', '--output', type=str, default=" ",
                    help='file to save output to (svg)')
parser = common_args(parser)
args = parser.parse_args()


conn = create_connection(args.database)
if(args.listtables):
    tables = list_tables(conn)
    for t in tables:
        print t
    exit(0)


# Set up global plotting properties
matplotlib.rcParams['figure.subplot.bottom'] = 0.2
matplotlib.rcParams['figure.subplot.top'] = 0.95
matplotlib.rcParams['figure.subplot.right'] = 0.98
matplotlib.rcParams['ytick.labelsize'] = 16
matplotlib.rcParams['xtick.labelsize'] = 16
matplotlib.rcParams['legend.fontsize'] = 18

# Get list of tables to compare
tables = list_tables(conn)
print args.tables
try:
    if set(args.tables) & set(tables) == set(args.tables):
        tables=args.tables
    else:
        print 'sorry couldnt find all of your tables, try a --
            listtables'
        exit(0)
except TypeError:
    print 'Empty list of tables provided, plotting all'

M = len(tables)

# Colors to distinguish each table in the bar graph
# More than 5 tables will wrap around to the start.
colors = ['b', 'r', 'g', 'm', 'k']

# Set up figure for plotting
f0 = plt.figure(0, facecolor='w', figsize=(14,10))
s0 = f0.add_subplot(1,1,1)

# Create a register of names that exist in all tables
tmp_regs = []
for table in tables:
    # Get results from the next table
    res = get_results(conn, table)

    tmp_regs.append(list())
    for r in res:
        try:
            tmp_regs[-1].index(r['kernel'])
```

```
            except ValueError:
                tmp_regs[-1].append(r['kernel'])

    # Get only those names that are common in all tables
    name_reg = tmp_regs[0]
    for t in tmp_regs[1:]:
        name_reg = list(set(name_reg) & set(t))
    name_reg.sort()

    # Pull the data out for each table into a dictionary
    # we can ref the table by it's name and the data associated
    # with a given kernel in name_reg by it's name.
    # This ensures there is no sorting issue with the data in the
    # dictionary, so the kernels are plotted against each other.
    table_data = collections.OrderedDict()
    for i,table in enumerate(tables):
        # Get results from the next table
        print 'results from '+table
        res = get_results(conn, table)

        data = dict()
        for r in res:
            data[r['kernel']] = r

        table_data[table] = data

    if args.percent is not None:
        for i,t in enumerate(table_data):
            if args.percent == t:
                norm_data = []
                for name in name_reg:
                    if(args.plot == 'max'):
                        norm_data.append(table_data[t][name]['max'])
                    elif(args.plot == 'min'):
                        norm_data.append(table_data[t][name]['min'])
                    elif(args.plot == 'mean'):
                        norm_data.append(table_data[t][name]['avg'])


    # Plot the results
    x0 = xrange(len(name_reg))
    i = 0
    # put in to an ordered dict (and order by key) so similar tables
        come out
    # matching colors -- could probably be improved by using
        orderdict to start
#    table_data = collections.OrderedDict(sorted(table_data.items(),
    key=lambda t: t[0]))
    for t in (table_data):
        ydata = []
        stds = []
        for name in name_reg:
            stds.append(math.sqrt(table_data[t][name]['var']))
            if(args.plot == 'max'):
                ydata.append(table_data[t][name]['max'])
            elif(args.plot == 'min'):
                ydata.append(table_data[t][name]['min'])
```

```
            elif(args.plot == 'mean'):
                ydata.append(table_data[t][name]['avg'])

        if args.percent is not None:
            ydata = [-100*(y-n)/y for y,n in zip(ydata,norm_data)]
            if(args.percent != t):
                # makes x values for this data set placement
                # width of bars depends on number of comparisons
                wdth = 0.80/(M-1)
                x1 = [x + i*wdth for x in x0]
                i += 1

                s0.bar(x1, ydata, width=wdth,
                       color=colors[(i-1)%M], label=t,
                       edgecolor='k', linewidth=2)

        else:
            # makes x values for this data set placement
            # width of bars depends on number of comparisons
            wdth = 0.80/M
            x1 = [x + i*wdth for x in x0]
            i += 1

            if(args.errorbars is False):
                s0.bar(x1, ydata, width=wdth,
                       color=colors[(i-1)%M], label=t,
                       edgecolor='k', linewidth=2)
            else:
                s0.bar(x1, ydata, width=wdth,
                       yerr=stds,
                       color=colors[i%M], label=t,
                       edgecolor='k', linewidth=2,
                       error_kw={"ecolor": 'k', "capsize":5,
                                 "linewidth":2})

    nitems = res[0]['nsamples']
    if args.percent is None:
        s0.set_ylabel("Processing time (sec) [{0:G} items]".format(
            nitems),
                      fontsize=22, fontweight='bold',
                      horizontalalignment='center')
    else:
        s0.set_ylabel("% Improvement over {0} [{1:G} items]".format(
                args.percent, nitems),
                      fontsize=22, fontweight='bold')

    s0.legend()
    s0.set_xticks(x0)
    s0.set_xticklabels(name_reg)
    for label in s0.xaxis.get_ticklabels():
        label.set_rotation(45)
        label.set_fontsize(16)

    if args.output == " ":
        plt.show()
    else:
        plt.savefig(args.output, format='pdf')
```

```
201 if __name__ == "__main__":
        main()
```

Listing 4: MatPlotLib plotting tools for mp-sched results.

```
    #!/usr/bin/python
  2 #!/usr/bin/env python

    import sys, math
    import argparse
  6 from common_test_funcs import *
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib import cm

 10 try:
        import matplotlib
        import matplotlib.pyplot as plt
    except ImportError:
 14     sys.stderr.write("Could␣not␣import␣Matplotlib␣(http://matplotlib.
            sourceforge.net/)\n")
        sys.exit(1)

    def main():
 18     desc='Plot␣synthetic␣FFT/FIR␣filters␣from␣a␣pickled␣file␣of␣results
            .␣' + \
             'Run␣one␣of␣the␣synthetic␣tests␣first␣(preferbly␣with␣
                 run_benchmarking)'
        parser = argparse.ArgumentParser(description=desc)
        parser.add_argument('-T', '--tables', type=str, nargs='*',
 22                         default=None,
                            help='select␣the␣tables␣to␣plot')
        parser.add_argument('-o', '--output', type=str, default="␣",
                            help='file␣to␣save␣output␣to␣(svg)')
 26

        parser = common_args(parser)
        args = parser.parse_args()
 30     conn = create_connection(args.database)

        tables = list_tables(conn)
        if(args.listtables):
 34         for t in tables:
                print t
            exit(0)


 38

        # Set up global plotting properties
        matplotlib.rcParams['figure.subplot.bottom'] = 0.2
        matplotlib.rcParams['figure.subplot.top'] = 0.95
 42     matplotlib.rcParams['figure.subplot.right'] = 0.98
        matplotlib.rcParams['ytick.labelsize'] = 16
        matplotlib.rcParams['xtick.labelsize'] = 16
        matplotlib.rcParams['legend.fontsize'] = 18
 46

        # Get list of tables to compare
        tables = list_tables(conn)
```

```
    try:
        if tables.__contains__(args.tables[0]):
            tables=args.tables[0]
        else:
            print 'sorry couldnt find all of your tables, try a --
                listtables'
            exit(0)
    except TypeError:
        print 'Empty list of tables provided!'
        exit(1)

#    f0 = plt.figure(0, facecolor='w', figsize=(14,14))
#    s0 = f0.add_subplot(1,1,1)
    f1 = plt.figure(1, facecolor='1', figsize=(14,14))
    s1 = f1.add_subplot(111, projection='3d', azim=230)


    res = get_results(conn, tables)

    max_stages = 0
    max_pipes = 0
    min_stages = 20
    min_pipes = 20
    results = numpy.zeros((10,10))
    for r in res:
        max_stages = max(max_stages, r['stages'])
        min_stages = min(max_stages, r['stages'])
        max_pipes = max(max_pipes, r['pipes'])
        min_pipes = min(max_pipes, r['pipes'])
        results[r['pipes']-1,r['stages']-1] = r['pseudoflopreal']

#    stages = numpy.arange(min_stages, max_stages)
#    pipes = numpy.arange(min_stages, max_stages)
    stages = numpy.arange(1,11)
    pipes = numpy.arange(1,11)
    pipes3d, stages3d = numpy.meshgrid(pipes, stages)
#    print results.size
#    print pipes.size
#    print stages.size
    surf = s1.plot_surface(pipes3d, stages3d, results, rstride=1,
        cstride=1,cmap=cm.jet,
                           linewidth=0, antialiased=True)
#    f1.colorbar(surf, shrink=0.5, aspect=5)
#    s1.set_zlim3d(results[min_pipes-1, min_stages-1]/2, results.max()
    *1.05)
#    contour_plot = s0.contour(pipes,stages,results/(10**9))
    plt.title('GFLOP/sec', fontsize=28)
    plt.xlabel('# pipes', fontsize=20)
    plt.ylabel('# stages', fontsize=20)
#    plt.clabel(contour_plot, inline=1, fontsize=14)

    if args.output == " ":
        plt.show()
    else:
        plt.savefig(args.output, format='pdf')

```

```
106 if __name__ == "__main__":
        main()
```

Listing 5: Modified mp-sched testing for FFTs

```
  1 #!/usr/bin/python
    #!/usr/bin/env python
    #
    # Copyright 2008 Free Software Foundation, Inc.
  5 #
    # This file is part of GNU Radio
    #
    # GNU Radio is free software; you can redistribute it and/or modify
  9 # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation; either version 3, or (at your option)
    # any later version.
    #
 13 # GNU Radio is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    # GNU General Public License for more details.
 17 #
    # You should have received a copy of the GNU General Public License
        along
    # with this program; if not, write to the Free Software Foundation, Inc
        .,
    # 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 21 #

    """
    Run synthetic.py for npipes in [1,16], nstages in [1,16]
 25 """

    import re
    import sys
 29 import os
    import tempfile
    from optparse import OptionParser
    from common_test_funcs import *
 33
    current_dir = os.path.dirname(__file__)

    def write_shell_script(f, description, ncores, gflops,
        max_pipes_and_stages, database):
 37     """
        f is the file to write the script to
        data_filename is the where the data ends up
        description describes the machine
 41     ncores is the number of cores (used to size the workload)
        gflops is the estimated GFLOPS per core (used to size the workload)
        """

 45     f.write("#!/bin/sh\n")
        f.write("(\n")
        if description:
            f.write("echo␣'#D␣%s'\n" % (description,))
```

```
49
        for npipes in range(1, max_pipes_and_stages + 1):
            for nstages in range(1, max_pipes_and_stages + 1):
                # We'd like each run of synthetic to take ~10 seconds
53              desired_time_per_run = 10
                est_gflops_avail = min(nstages * npipes, ncores) * gflops
                nsamples = (est_gflops_avail * desired_time_per_run)/(512.0
                     * nstages * npipes)
                nsamples = int(nsamples * 1e9)
57
                cmd = "./%s/synthetic_fft.py -m -s %d -p %d -N %d -D %s\n"
                    % (current_dir, nstages, npipes, nsamples, database)
                f.write(cmd)
                f.write('if test $? -ge 128; then exit 128; fi\n')
61
        f.write(") 2>&1 \n" )
        f.flush()


65


    def main():
        description = """%prog gathers multiprocessor scaling data using
            the ./synthetic.py benchmark.
69  All combinations of npipes and nstages between 1 and --max-pipes-and-
        stages are tried.
    The -n and -f options provides hints used to size the workload.  We'd
        like each run
    of synthetic to take about 10 seconds.  For the full 16x16 case this
        results in a
    total runtime of about 43 minutes, assuming that your values for -n and
         -f are reasonable.
73  For x86 machines, assume 3 FLOPS per processor Hz. E.g., 3 GHz machine
        -> 9 GFLOPS.
    plot_flops.py will make pretty graphs from the output data generated by
         %prog.
    """
        usage = "usage: %prog [options] -D results.db -L label"
77      parser = OptionParser(usage=usage, description=description)
        parser.add_option("-d", "--description", metavar="DESC",
                          help="machine description, e.g., \"Dual quad-core
                              Xeon 3 GHz\"", default=None)
        parser.add_option("-n", "--ncores", type="int", default=1,
81                        help="number of processor cores [default=%default
                              ]")
        parser.add_option("-g", "--gflops", metavar="GFLOPS", type="float",
             default=3.0,
                          help="estimated GFLOPS per core [default=%default
                              ]")
        parser.add_option("-m", "--max-pipes-and-stages", metavar="MAX",
            type="int", default=16,
85                        help="maximum number of pipes and stages to use [
                              default=%default]")
        parser.add_option("-D", "--database", metavar="CONN", type="str")
        parser.add_option("-L", "--table", metavar="TABLE", type="str")

89      (options, args) = parser.parse_args()
```

```
        shell = os.popen("/bin/sh", "w")
93
        write_shell_script(shell,
                           options.description,
                           options.ncores,
97                         options.gflops,
                           options.max_pipes_and_stages,
                           options.database)
        c=create_connection(options.database)
101     new_table(c, options.table)
        close_connection(c)

    if __name__ == '__main__':
105     main()
```